

Process Management

LinuXpert Systems

No.5, Dr. Ambedkar St, Rathinapuram,

Nanganallur, Chennai – 600 061.

Phone: 98841 65649, 90808 01382

E-mail: baskar@linuxpert.in

TABLE OF CONTENTS

1. ps 1

2. top 4

3. htop 10

4. kill 16

5. pkill 21

6. killall 27

7. nice 32

8. renice 36

9. jobs 40

10. bg 46

11. fg 50

1. ps

The **ps** command is a **process status** command in Linux/Unix systems. It is used to **display information about currently running processes**.

Basic usage

```
ps
```

Running `ps` without options shows processes running in your **current terminal session**.

Example output:

```
  PID TTY          TIME CMD
 1234 pts/0    00:00:01 bash
 5678 pts/0    00:00:00 ps
```

Columns explained

Column	Meaning
`PID`	Process ID — a unique number assigned to each running process
`TTY`	Terminal associated with the process
`TIME`	CPU time used by the process
`CMD`	Command that started the process

Common `ps` options

1. Show all processes for your user

```
ps -x
```

Shows processes running even without a terminal (such as background processes).

---

2. Show all processes on the system

```
ps -e
```

or:

```
ps -A
```

Example:

```
PID   TTY      TIME     CMD
1     ?        00:00:05 systemd
250   ?        00:00:01 sshd
1024  pts/0    00:00:00 bash
```

---

3. Detailed process information

```
ps -f
```

The `-f` option gives a full-format listing:

```
UID   PID  PPID  C STIME TTY      TIME CMD
user 1234 1000  0 10:30 pts/0    00:00 bash
```

Additional columns:

- `UID` → user who owns the process
- `PPID` → parent process ID
- `STIME` → start time

4. The most commonly used version

```
ps aux
```

This displays **all running processes with detailed information**.

Example:

```
USER   PID %CPU %MEM    VSZ   RSS TTY STAT START CMD
root     1  0.0  0.1 170000 10000 ?   Ss   10:00 systemd
john  2345  5.2  2.0 500000 80000 ?   Sl   10:15 firefox
```

Meaning:

- `%CPU` → CPU usage
- `%MEM` → memory usage
- `VSZ` → virtual memory used
- `RSS` → physical memory used
- `STAT` → process state

---

5. Find a specific process

Example: find Firefox:

```
ps aux | grep firefox
```

The `|` (pipe) sends the output of `ps aux` to `grep`, which searches for matching text.

Related commands

Command	Purpose
`top`	Live view of running processes
`htop`	Interactive version of `top`
`kill PID`	Stop a process by PID
`pgrep name`	Find PID by process name

In short:

```
ps
```

= **"Show me the processes running on this system/session."**

2. top

The **`top`** command is a **real-time system monitoring command** in Linux/Unix. It displays **running processes, CPU usage, memory usage, and system load information** in a continuously updating interface.

Think of `top` as a **live dashboard for your computer's activity**.

Basic usage

```
top
```

Example output:

```
top - 10:30:15 up 2 days,  3:45,  1 user,  load average: 0.20, 0.15, 0.10
Tasks: 150 total,   2 running, 148 sleeping
%Cpu(s):  5.0 us,  1.0 sy, 94.0 id
MiB Mem :  16000 total,  8000 free,  5000 used

PID   USER     %CPU  %MEM   COMMAND
1234  user      25.0   5.0   firefox
5678  root       2.0   1.0   sshd
```

Understanding the top screen

1. System summary

Example:

```
top - 10:30:15 up 2 days, 3:45
```

- `10:30:15` → current system time
- `up 2 days` → how long the system has been running (uptime)

2. Load average

Example:

```
load average: 0.20, 0.15, 0.10
```

Shows system workload:

- First value → last 1 minute
- Second value → last 5 minutes
- Third value → last 15 minutes

Lower values usually mean less CPU pressure.

For a 4-core CPU:

- Load `1.0` → roughly one CPU core fully busy
- Load `4.0` → all cores fully busy

---

3. Task/process summary

Example:

```
Tasks: 150 total, 2 running, 148 sleeping
```

Shows:

- Total number of processes
- Running processes
- Sleeping/idle processes

4. CPU usage

Example:

```
%Cpu(s): 5.0 us, 1.0 sy, 94.0 id
```

Field	Meaning
us	User processes (applications)
sy	System/kernel processes
ni	Processes with modified priority
id	Idle CPU
wa	Waiting for disk/network I/O
hi	Hardware interrupts
si	Software interrupts

Example:

```
95.0 id
```

means the CPU is mostly idle.

---

5. Memory usage

Example:

```
MiB Mem : 16000 total, 8000 free, 5000 used
```

Shows RAM usage:

Field	Meaning
Total	Total RAM available
Free	Unused RAM
Used	RAM currently in use

Process list columns

Example:

```
PID USER %CPU %MEM COMMAND
```

Column	Meaning
`PID`	Process ID
`USER`	Owner of process
`%CPU`	CPU usage percentage
`%MEM`	Memory usage percentage
`COMMAND`	Program name

Useful keys while running `top`

Sort by CPU usage

Press:

```
Shift + P
```

Shows processes using the most CPU.

Sort by memory usage

Press:

```
Shift + M
```

Shows processes using the most RAM.

Kill a process

Press:

```
k
```

Then enter the PID.

Example:

```
k
1234
```

This sends a termination signal to process `1234`.

Change refresh speed

Press:

```
d
```

Then enter seconds:

```
5
```

Updates every 5 seconds.

Exit top

Press:

```
q
```

Common alternatives

`top` with better interface

```
htop
```

`htop` provides a more interactive, colour-based view.

Show only a specific user's processes

```
top -u username
```

Example:

```
top -u john
```

Batch mode (useful for scripts)

```
top -b -n 1
```

- `-b` → batch output
- `-n 1` → take one snapshot

Difference between `ps` and `top`

`ps`	`top`
Gives a snapshot	Continuously updates
Shows processes once	Shows live activity
Good for searching processes	Good for monitoring
Example: `ps aux`	Example: `top`

In short:

```
top
```

= **"Show me a live view of what my system is doing right now."**

3. htop

The **`htop`** command is an **interactive process monitoring tool** for Linux/Unix systems. It is an improved, more user-friendly alternative to `top`.

It shows **running processes, CPU usage, memory usage, and system load in real time** with a colour-based interface and allows you to manage processes directly.

Basic usage

```
htop
```

Example view:

```
  CPU[||||||||         25%]   Tasks: 120, 3 running
  Mem[|||||||          40%]   Load average: 0.50 0.30 0.20

  PID   USER     CPU%  MEM%   TIME   COMMAND
 1234   user     25.0  5.2    2:10   firefox
 5678   root      3.0  1.0    0:20   sshd
```

Main sections of `htop`

1. CPU usage

At the top:

```
CPU[||||||||     25%]
```

Shows CPU usage.

For multi-core systems, you may see:

```
CPU 1 [||||      20%]
CPU 2 [||||||||  50%]
CPU 3 [||        10%]
```

Each bar represents one CPU core.

2. Memory usage

Example:

```
Mem[||||||||      40%]
```

Shows RAM usage.

- Filled area → used memory
- Empty area → available memory

---

3. Swap usage

Example:

```
Swp[||            5%]
```

Shows swap memory usage (disk space used as extra memory).

Process list

Example:

```
PID   USER     CPU%  MEM%  COMMAND
```

Column	Meaning
PID	Process ID
USER	User running the process
CPU%	CPU usage
MEM%	Memory usage
TIME	CPU time consumed
COMMAND	Program name

Useful keyboard shortcuts in `htop`

Sort processes

Press:

`F6`

Then choose:

- CPU%
- MEM%
- PID
- USER
- TIME

Example:

- Sort by CPU → find programs using the most processor power
- Sort by MEM → find programs using the most RAM

Kill a process

Select a process and press:

`F9`

Choose a signal:

Common options:

- `SIGTERM` → politely ask the process to stop
- `SIGKILL` → forcefully stop the process

Search for a process

Press:

`F3`

Example:

`firefox`

It highlights matching processes.

Filter processes

Press:

`F4`

Example:

`python`

Shows only Python processes.

Tree view

Press:

`F5`

Displays parent-child relationships:

```
systemd
 ├── sshd
 │    └── bash
 └── nginx
```

Useful for understanding which process started another process.

Change refresh interval

Press:

`F2`

Go to settings → adjust update delay.

Quit `htop`

Press:

`F10`

or:

`q`

Installing `htop`

Some Linux distributions do not include it by default.

Ubuntu/Debian

```
sudo apt install htop
```

Fedora/RHEL

```
sudo dnf install htop
```

Arch Linux

```
sudo pacman -S htop
```

`top` vs `htop`

Feature	`top`	`htop`
Interface	Basic text	Colourful interactive UI
Mouse support	No	Yes (usually)
Process selection	Keyboard only	Easy selection
Kill processes	Manual PID entry	Select and kill
Tree view	Limited	Built-in
Sorting	Commands	Menu-based

In short:

`htop`

means:

"Open an interactive dashboard where I can watch system performance and manage running processes easily."

4. kill

The **`kill`** command is a **Linux/Unix command used to send signals to running processes**. It is commonly used to **stop, restart, or control a process** by using its **Process ID (PID)**.

Despite its name, `kill` does not always terminate a process. It sends a signal, and the process decides how to respond (except for signals like `SIGKILL` that cannot be ignored).

Basic syntax

```
kill [signal] PID
```

Example:

```
kill 1234
```

This sends the default termination signal to the process with PID `1234`.

Finding a PID

Use:

```
ps aux
```

Example:

```
USER   PID   %CPU  COMMAND
john   1234  20.0  firefox
root   5678   1.0  sshd
```

Here:

- Firefox PID = `1234`
- SSH daemon PID = `5678`

You can stop Firefox with:

```
kill 1234
```

Common signals

1. SIGTERM (15) — graceful termination

Default signal:

```
kill PID
```

or:

```
kill -15 PID
```

Meaning:

> "Please stop gracefully."

The process gets a chance to:

- save data
- close files
- clean up resources

Example:

```
kill 1234
```

This is the recommended way to stop most programs.

---

2. SIGKILL (9) — force stop

```
kill -9 PID
```

Meaning:

> "Stop immediately."

The process cannot ignore this signal.

Example:

```
kill -9 1234
```

Use when:

- a program is frozen
- `kill PID` does not work
- a process refuses to exit

It may cause data loss because the program cannot perform cleanup.

---

3. SIGHUP (1) — reload configuration

```
kill -HUP PID
```

Often used by services to reload their configuration without restarting.

Example:

```
kill -HUP 5678
```

---

4. SIGSTOP — pause a process

```
kill -STOP PID
```

Freezes a process.

Example:

```
kill -STOP 1234
```

The process remains in memory but does not run.

---

5. SIGCONT — resume a paused process

```
kill -CONT PID
```

Restarts a process stopped with `SIGSTOP`.

Example:

```
kill -CONT 1234
```

List all available signals

```
kill -l
```

Example output:

```
1) SIGHUP
2) SIGINT
9) SIGKILL
15) SIGTERM
18) SIGCONT
19) SIGSTOP
```

Killing multiple processes

Using `pkill`:

```
pkill firefox
```

Stops processes matching the name `firefox`.

Using `killall`:

```
killall firefox
```

Stops all processes with that name.

Kill a background job

Example:

Start a program:

```
firefox &
```

Output:

```
[1] 1234
```

The number `1234` is the PID.

Stop it:

```
kill 1234
```

Using `kill` with `top` or `htop`

In `top`:

1. Press:

```
k
```

2. Enter PID:

```
1234
```

3. Choose the signal.

In `htop`:

1. Select the process
2. Press:

```
F9
```

3. Choose the signal

Difference between related commands

Command	Purpose
`kill PID`	Send a signal to a process
`kill -9 PID`	Forcefully terminate a process
`pkill name`	Kill processes by name
`killall name`	Kill all processes with a name
`systemctl stop service`	Stop a system service

In short:

```
kill PID
```

means:

"Send a request to a specific running process to perform an action, usually to stop it."

5. pkill

The `pkill` command is a **Linux/Unix command used to send signals to processes based on their name or other attributes**, instead of requiring the process ID (PID).

It is similar to `kill`, but instead of doing:

```
kill PID
```

you can do:

```
pkill process_name
```

and `pkill` finds the matching process(es) automatically.

Basic syntax

```
pkill [options] process_name
```

Example:

```
pkill firefox
```

This sends the default **SIGTERM** signal to all processes named `firefox`.

How `pkill` works

Suppose you run:
`ps aux | grep firefox`

Output:

```
john   1234  20.0  firefox
john   5678   5.0  firefox
```

Instead of:

```
kill 1234
kill 5678
```

you can simply run:

```
pkill firefox
```

Both Firefox processes will receive the termination signal.

Common `pkill` options

1. Kill a process by name

```
pkill nginx
```

Stops all processes named `nginx`.

---

2. Force kill a process

```
pkill -9 firefox
```

Equivalent to:

```
kill -9 PID
```

It sends `SIGKILL`, which immediately terminates the process.

Use this when a process is frozen and does not respond to normal termination.

---

3. Show what would be killed

```
pkill -0 firefox
```

This checks whether matching processes exist without actually killing them.

---

4. Kill processes owned by a user

```
pkill -u username
```

Example:

```
pkill -u john
```

Terminates all processes belonging to user `john`.

5. Match the full command name

By default, `pkill` matches the process name.

Example:

```
pkill python
```

May match:

```
python
python3
```

To match the exact name:

```
pkill -x python
```

Only processes exactly named `python` are affected.

---

6. Send a specific signal

Syntax:

```
pkill -SIGNAL process_name
```

Example:

```
pkill -HUP nginx
```

Sends `SIGHUP` to nginx, commonly used to reload configuration.

Example:

```
pkill -STOP firefox
```

Pauses Firefox.

Resume:

```
pkill -CONT firefox
```

7. Ask before killing

```
pkill -i firefox
```

Interactively asks for confirmation before sending the signal.

Finding processes before using `pkill`

You can check matching processes with:

```
pgrep firefox
```

Example:

```
1234
5678
```

Then:

```
pkill firefox
```

Difference between `kill`, `pkill`, and `killall`

Command	Works with	Example
`kill`	PID	`kill 1234`
`pkill`	Process name/pattern	`pkill firefox`
`killall`	Exact process name	`killall firefox`

Practical examples

Stop Firefox

```
pkill firefox
```

Force close a stuck application

```
pkill -9 application_name
```

Example:

```
pkill -9 chrome
```

Reload a service configuration

```
pkill -HUP nginx
```

Stop all Python processes

```
pkill python
```

Important caution

`pkill` can affect **multiple processes at once**. For example:

```
pkill java
```

may stop every Java program running on the machine.

Before using it on important systems, check what matches:

```
pgrep -a java
```

In short:

```
pkill process_name
```

means:

"Find processes matching this name and send them a signal."

It is a faster way to control processes when you do not know (or do not want to look up) their PIDs.

6. killall

The **`killall`** command is a **Linux/Unix command used to terminate processes by their name**. It sends a signal to **all processes that match a specified name**.

Unlike `kill`, which requires a **PID**, `killall` works with the **process name**.

Basic syntax

```
killall [options] process_name
```

Example:

```
killall firefox
```

This sends the default termination signal (`SIGTERM`) to all running processes named `firefox`.

How `killall` works

Suppose you run:

```
ps aux | grep firefox
```

Output:

```
john   1234  15.0  firefox
john   5678   8.0  firefox
```

Using `kill`:

```
kill 1234
kill 5678
```

Using `killall`:

```
killall firefox
```

Both Firefox processes receive the termination signal.

Common `killall` options

1. Kill processes by name

```
killall nginx
```

Stops all processes named `nginx`.

---

2. Force kill a process

```
killall -9 firefox
```

Sends `SIGKILL`.

Difference:

- Normal:

```
killall firefox
```

Allows the application to clean up.

- Force:

```
killall -9 firefox
```

Stops immediately.

Use `-9` only when a process is stuck.

---

3. Ask for confirmation

```
killall -i firefox
```

Example:

```
Kill firefox(1234)? (y/N)
```

Useful to avoid accidentally stopping the wrong processes.

4. Show what would be killed

```
killall -v firefox
```

Displays information about the processes being terminated.

Example:

```
Killed firefox(1234)
Killed firefox(5678)
```

---

5. Kill processes belonging to a user

```
killall -u username process_name
```

Example:

```
killall -u john firefox
```

Only kills Firefox processes owned by user `john`.

---

6. Kill processes older/newer than a time

Kill processes older than a certain time:

```
killall -o 1h firefox
```

Meaning:

- `-o` → older than
- `1h` → one hour

Kill processes younger than a time:

```
killall -y 10m firefox
```

Meaning:

- `-y` → younger than
- `10m` → ten minutes

Difference between `kill`, `pkill`, and `killall`

Command	Matches by	Example
`kill`	PID	`kill 1234`
`pkill`	Name/pattern	`pkill firefox`
`killall`	Exact process name	`killall firefox`

Examples

Close Firefox

```
killall firefox
```

Force close a frozen application

```
killall -9 chrome
```

Stop all instances of a program

```
killall python
```

This stops every process named `python`.

Important caution

`killall` can terminate **multiple processes at once**.
For example:

```
killall java
```

may stop every Java application running on the system.

Before using it on a production machine, check matching processes first:

```
pgrep -a java
```

In short:

`m`eans:

"Find every running process with this name and send it a signal."

It is useful for quickly stopping all instances of an application, but it should be used carefully because it can affect many processes at once.

7. nice

The `nice` command is a **Linux/Unix command used to run a program with a modified CPU scheduling priority**.

It controls how much **CPU attention** a process receives compared with other processes.

A process with a **higher niceness value** gets **lower CPU priority**, while a process with a **lower niceness value** gets **higher CPU priority**.

Basic syntax

```
nice [option] command
```

Example:

```
nice firefox
```

This starts Firefox with the default adjusted priority.

Understanding "nice value"

Linux processes have a **niceness value**:

- Range: **-20 to 19**
- Default: **0**

Nice value	Priority	Meaning
`-20`	Highest priority	Gets more CPU time
`0`	Normal priority	Default
`19`	Lowest priority	Gets less CPU time

Think of it as:

- **Low nice value → "not nice" to other processes → more CPU**
- **High nice value → "being nice" → gives CPU time to others**

Check process nice values

Use:

```
ps -el
```

Example:

```
F S UID  PID  NI CMD
4 S 1000 1234  0 firefox
4 S 1000 5678 10 backup
```

The `NI` column shows the nice value.

Run a command with a specific nice value

Use:
`nice -n value command`

Example:

```
nice -n 10 tar -czf backup.tar.gz /home/user
```

This runs the backup process with a nice value of `10`, giving it lower CPU priority.

Examples

Normal priority

```
id="w3n2xq"
nice -n 0 program
```

Runs with the default priority.

Lower CPU priority

```
id="x8q4pz"
nice -n 15 python script.py
```

Useful for:

- backups
- large data processing
- compression jobs

The system remains more responsive.

Increase priority (requires permission)

```
id="7jz6kf"
sudo nice -n -10 program
```

Runs with higher CPU priority.

Usually only root users can set negative nice values.

`nice` vs `renice`

Command	Purpose	Example
`nice`	Start a new process with a priority	`nice -n 10 program`
`renice`	Change priority of an existing process	`renice 10 -p 1234`

Example workflow

Start a CPU-heavy job with low priority:

```
nice -n 19 ./compile.sh
```

Find its PID:

```
ps aux | grep compile
```

Change its priority later:

```
renice 5 -p 1234
```

Important notes

- A higher nice value **does not make a program slower directly**. It only means the scheduler gives CPU time to other processes first.
- It affects **CPU scheduling only**, not memory or disk priority.
- A process with nice value `19` can still use 100% CPU if no other processes need it.

In short:

```
nice -n 10 command
```

means:

"Run this command with lower CPU priority so other processes get preference."

It is commonly used for background tasks like backups, compilation, video encoding, and large data processing jobs.

8. renice

The `renice` command is a **Linux/Unix command used to change the CPU scheduling priority (nice value) of an already running process**.

While `nice` sets the priority **when starting a new process**, `renice` changes the priority of a **process that is already running**.

Basic syntax

```
renice [priority] -p PID
```

Example:

```
renice 10 -p 1234
```

This changes the process with PID `1234` to a nice value of `10`.

Understanding nice values

Linux uses a **nice value** to decide CPU scheduling priority.

Range:

```
-20  ← Highest CPU priority
  0  ← Default priority
 19  ← Lowest CPU priority
```

Nice value	CPU priority	Example use
`-20`	Highest	Critical system tasks
`0`	Normal	Regular applications
`10`	Lower	Background jobs
`19`	Lowest	Very low priority tasks

Remember:

- **Lower nice value = more CPU priority**
- **Higher nice value = less CPU priority**

Find a process PID

Before using `renice`, find the PID:

```
ps aux
```

Example:

```
USER   PID   %CPU   COMMAND
john   1234  80.0   blender
john   5678   5.0   firefox
```

Here:

- Blender PID = `1234`

Change process priority

Lower CPU priority

```
renice 10 -p 1234
```

Output:

```
1234: old priority 0, new priority 10
```

Now the process gets less CPU preference.

Set very low priority

```
renice 19 -p 1234
```

Useful for:

- backups
- large downloads
- video encoding
- data processing

Increase CPU priority

```
sudo renice -10 -p 1234
```

Negative nice values require administrator privileges.

This gives the process more CPU preference.

Change priority for a user

You can change the priority of all processes owned by a user:

```
renice 10 -u username
```

Example:

```
renice 15 -u john
```

All processes owned by `john` get a nice value of `15`.

Change priority of a process group

```
renice 10 -g group_id
```

Changes the priority of all processes in a process group.

Check the new priority

Use:

```
ps -o pid,ni,cmd -p PID
```

Example:

```
ps -o pid,ni,cmd -p 1234
```

Output:

```
PID   NI   CMD
1234  10   blender
```

`NI` shows the current nice value.

Difference between `nice` and `renice`

Command	When used	Example
`nice`	Start a new process with a priority	`nice -n 10 ./backup.sh`
`renice`	Change priority of an existing process	`renice 10 -p 1234`

Practical examples

Reduce CPU usage of a running compilation

Find the compiler PID:

```
ps aux | grep gcc
```

Change priority:

```
renice 15 -p 4567
```

Give a video encoder more CPU priority

```
sudo renice -5 -p 7890
```

Important notes

- `renice` does **not stop or restart** a process.
- It only changes how the CPU scheduler treats the process.
- Regular users can usually increase nice values (lower priority), but only privileged users can decrease nice values (increase priority).
- It affects CPU scheduling, not RAM or disk usage.

In short:

```
renice value -p PID
```

means:

"Change the CPU priority of an already running process."

9. jobs

The `jobs` command is a **Linux/Unix shell command used to display the background and stopped jobs running in your current terminal session**.

It is mainly used for **job control** — managing programs that you started from the command line using `&`, `Ctrl+Z`, `bg`, and `fg`.

Basic syntax

```
jobs
```

Example:

```
$ jobs
[1]   Running    firefox &
[2]   Stopped    vim
```

Understanding the output

Example:

```
[1]   Running    firefox &
[2]   Stopped    vim
```

Part	Meaning
`[1]`	Job number assigned by the shell
`Running`	Current job state
`Stopped`	Job is paused
`firefox`	Command that started the job

The job number (`1`, `2`, etc.) is different from the process ID (PID).

How jobs are created

1. Running a command in the background

Normally:

```
firefox
```

The terminal waits until Firefox exits.

Run it in the background:

```
firefox &
```

Output:

```
[1] 12345
```

- `[1]` → job number
- `12345` → process ID (PID)

Now check:

```
jobs
```

Output:

```
[1]+ Running firefox &
```

---

2. Stopping a running job

Start a program:

```
vim
```

Press:

```
Ctrl + Z
```

Output:

```
[1]+ Stopped vim
```

Check:

```
jobs
```

Output:

```
[1]+ Stopped vim
```

The process is paused but still exists.

Useful `jobs` options

1. Show process IDs

```
jobs -l
```

Example:

```
[1]+ 12345 Running firefox &
[2]- 12346 Stopped vim
```

Here:

- `12345` and `12346` are PIDs

---

2. Show only running jobs

```
jobs -r
```

Example:

```
[1]+ Running firefox &
```

---

3. Show only stopped jobs

```
jobs -s
```

Example:

```
[2]+ Stopped vim
```

Related job control commands

`bg` — move a stopped job to background

Example:

```
bg %1
```

Meaning:

Continue job number 1 in the background.

Example:

```
[1]+ firefox &
```

`fg` — bring a background job to foreground

Example:

```
fg %1
```

Meaning:

Bring job number 1 back to the terminal.

`kill` — terminate a job

Example:

```
kill %1
```

Stops job number 1.

Job number vs PID

Example:

```
[1]+ 12345 Running firefox &
```

Value	Meaning
%1	Shell job number
12345	Process ID

Use:

```
kill %1
```

or:

```
kill 12345
```

Both can target the process.

Example workflow

Start a long task

```
./backup.sh &
```

Output:

```
[1] 25000
```

Check jobs

```
jobs
```

Output:

```
[1]+ Running ./backup.sh &
```

Bring it to foreground

```
fg %1
```

Pause it

Press:

`Ctrl + Z`

Continue it in background

```
Bg %1
```

Important notes

- `jobs` only shows jobs started from the **current shell session**.
- It will not show every running process on the system.
- For all system processes, use:

```
ps aux
```

or:

```
top
```

In short:

```
jobs
```

means:

"Show me the programs I started from this terminal that are running in the background or have been stopped."

10.bg

The **`bg`** command is a **Linux/Unix shell command used to continue a stopped job and run it in the background**.

It is part of **job control** and works with commands like `jobs`, `fg`, and `Ctrl+Z`.

Basic syntax

```
bg [job_number]
```

Example:

```
bg %1
```

This resumes job number `1` and runs it in the background.

How `bg` works

When you run a program normally:

```
vim
```

The terminal is occupied by Vim.

If you press:

```
Ctrl + Z
```

The shell pauses the program:

```
[1]+  Stopped  vim
```

Now check jobs:

```
jobs
```

Output:

```
[1]+ Stopped vim
```

The job is stopped, not running.

To continue it in the background:

```
bg %1
```

Output:

```
[1]+ vim &
```

Now Vim continues running, and you get your terminal back.

Job numbers

The % symbol refers to a shell job number.

Example:

```
jobs
```

Output:

```
[1]   Stopped   nano file.txt
[2]-  Stopped   python script.py
[3]+  Running   firefox &
```

Use:

```
bg %1
```

to resume job 1.

Using bg without a job number

```
bg
```

By default, it resumes the **most recently stopped job**.

Example:

```
[1]+ Stopped python script.py
```

Running:

```
bg
```

is equivalent to:

```
bg %1
```

Common workflow with job control

1. Start a program

```
python long_task.py
```

---

2. Pause it

Press:

```
Ctrl + Z
```

Output:

```
[1]+ Stopped python long_task.py
```

---

3. Move it to background

```
bg %1
```

Now it continues running in the background.

---

4. Check running jobs

```
jobs
```

Output:

```
[1]+ Running python long_task.py &
```

---

5. Bring it back to foreground

Use:

```
fg %1
```

Difference between `bg` and `fg`

Command	Purpose	Example
`bg`	Continue a stopped job in background	`bg %1`
`fg`	Bring a background job to foreground	`fg %1`

Difference between `bg` and `&`

Start a command in background:

```
command &
```

Example:

```
firefox &
```

The command starts in the background immediately.

Move an existing stopped command to background:

```
bg %1
```

The command was already running but paused with `Ctrl+Z`.

Important notes

- `bg` only works with jobs from the **current shell session**.
- It does not work with arbitrary system processes.
- Use `ps`, `top`, or `htop` to manage system-wide processes.

In short:

```
bg %1
```

means:

"Resume job number 1 and let it continue running in the background while I keep using the terminal."

11. fg

The `fg` command is a **Linux/Unix shell command used to bring a background job or a stopped job back to the foreground**.

It is part of **shell job control** and is commonly used with `jobs`, `bg`, and `Ctrl+Z`.

When a job is in the foreground, it **controls your terminal** — you can interact with it directly.

Basic syntax

```
fg [job_number]
```

Example:

```
fg %1
```

This brings job number `1` to the foreground.

How `fg` works

1. Start a program in the background

Example:

```
firefox &
```

Output:

```
[1] 12345
```

The & runs Firefox in the background.

Check jobs:

```
jobs
```

Output:

```
[1]+ Running firefox &
```

2. Bring it back to the foreground

Run:

```
fg %1
```

Now Firefox becomes the foreground job.

Your terminal is attached to that process again.

Using `fg` with stopped jobs

Suppose you run:

```
vim notes.txt
```

While Vim is running, press:

```
Ctrl + Z
```

The shell pauses it:

```
[1]+ Stopped vim notes.txt
```

Check:

```
jobs
```

Output:

```
[1]+ Stopped vim notes.txt
```

Bring it back:

```
fg %1
```

Vim resumes and takes control of the terminal again.

Job numbers

The % symbol indicates a shell job number.

Example:

```
jobs
```

Output:

```
[1]   Running    firefox &
[2]-  Stopped    nano file.txt
[3]+  Running    python app.py &
```

Bring job 2 to the foreground:

```
fg %2
```

Using `fg` without a job number

```
fg
```

This brings the **most recent job** (usually the one marked with + in `jobs`) to the foreground.

Example:

```
[1]- Running firefox &
[2]+ Stopped vim
```

Running:

```
fg
```

brings Vim (%2) back.

Common job-control workflow

Start a long-running task

```
python backup.py
```

Pause it

Press:

`Ctrl + Z`

Output:

```
[1]+ Stopped python backup.py
```

Continue in background

```
bg %1
```

Bring it back later

```
fg %1
```

Difference between `fg` and `bg`

Command	Purpose	Example
`fg`	Bring a job to the foreground	`fg %1`
`bg`	Continue a stopped job in the background	`bg %1`

Difference between `fg` and running a command normally

Normal:

```
vim file.txt
```

- Starts in foreground
- Terminal is occupied

Background:

```
vim file.txt &
```

- Starts in background
- Terminal remains available

Using:

```
fg %1
```

moves that background job back to the foreground.

Important notes

- `fg` only works with jobs started from the **current shell session**.
- It cannot bring arbitrary system processes to the foreground.
- To manage system processes, use tools like `ps`, `top`, or `htop`.

In short:

```
fg %1
```

means:

"Bring job number 1 back to the terminal and run it in the foreground."